|  |
| --- |
| Circle Language Spec: System Objects |

## System Objects

### Introduction

This chapter tries to demonstrate internal workings of symbols in Circle.

Circle Language Spec's aim might be to show the *notation* of Circle, but this chapter tries to look at the *internal* *workings* as if it were an actual program. In doing so, finer details of Circle might start to show. Specifically, usages of the *system interface* notation might come to light.

This hypothetical implementation supposes, that systems might be composed of *system objects*. System objects might be *actual* objects that systems would be running on.

One of the purposes of system objects could be to manage *relationships* between objects.

A system object may also manage aspects of a symbol, like its *class*, whether it *executes* or if it stores a *value*. These aspects might be controlled through *system commands*.

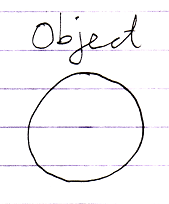
*Assignment commands* may also be system commands. An assignment command might copy an aspect from one object to another.

The *System Objects* documentation would also show *connectors, connections* and other notational forms that might come with controlling these system aspects.

### The System Objects

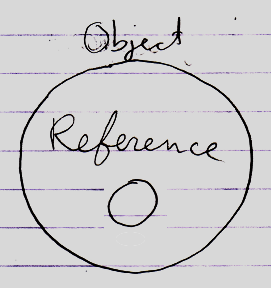
#### The Object

In object oriented programming, an *object* might be considered the most basic element of a computer program. An object might represent a thing, an idea or a place, a number or a collection of other things or possibly anything else. All those things might be called objects.



#### The Reference

An object might not be directly accessed. We might frequently be dealing with *references* to objects instead.



A reference might put an object into context.

Synonyms might be:

**Related Object**

**Related Item**

**Pointer**

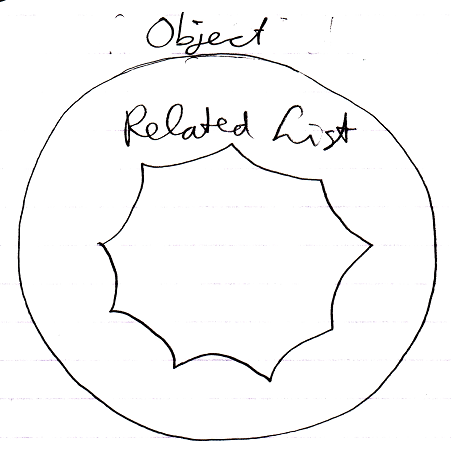
**Item**

**Reference**

**Object Reference**

#### Related List

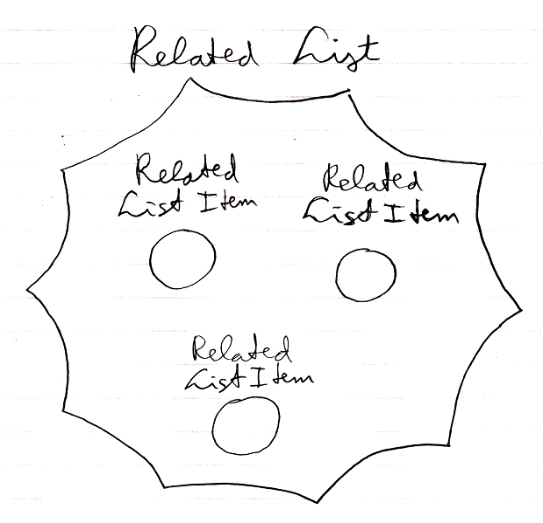
Next to related items, a parent object might also contain *related lists*.



A related list might also be a related object, but perhaps a special one that might contain a scalable collection of other objects.

#### Related List Item

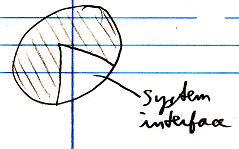
A related list may contain multiple *related list items*.



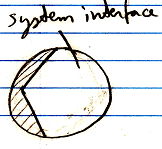
A related *list* itemmight be like a *related item*. What applies to related items might also apply to related *list* items. In some cases though, behavior may be different.

#### System Interface

To see how objects and references might work internally, we might crack open the 'system interface':



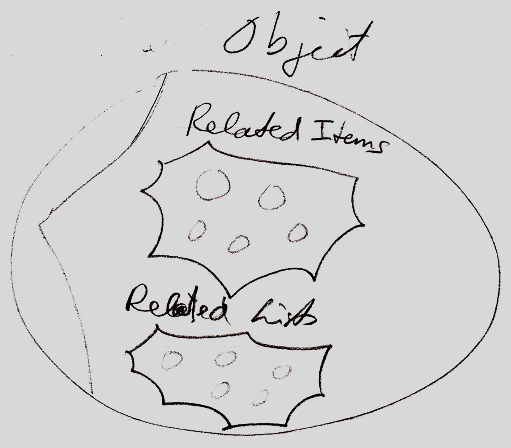
"and a little further"



More may follow about this system interface notation later.

#### Related Items & Related Lists Collections

Inside the system interface an object’s related items and related lists might be stored as the object’s containing two collections: **Related Items** and **Related Lists**. The **Related Items** collection would contain the related items of an object. The **Related Lists** collection would contain the related lists of an object.

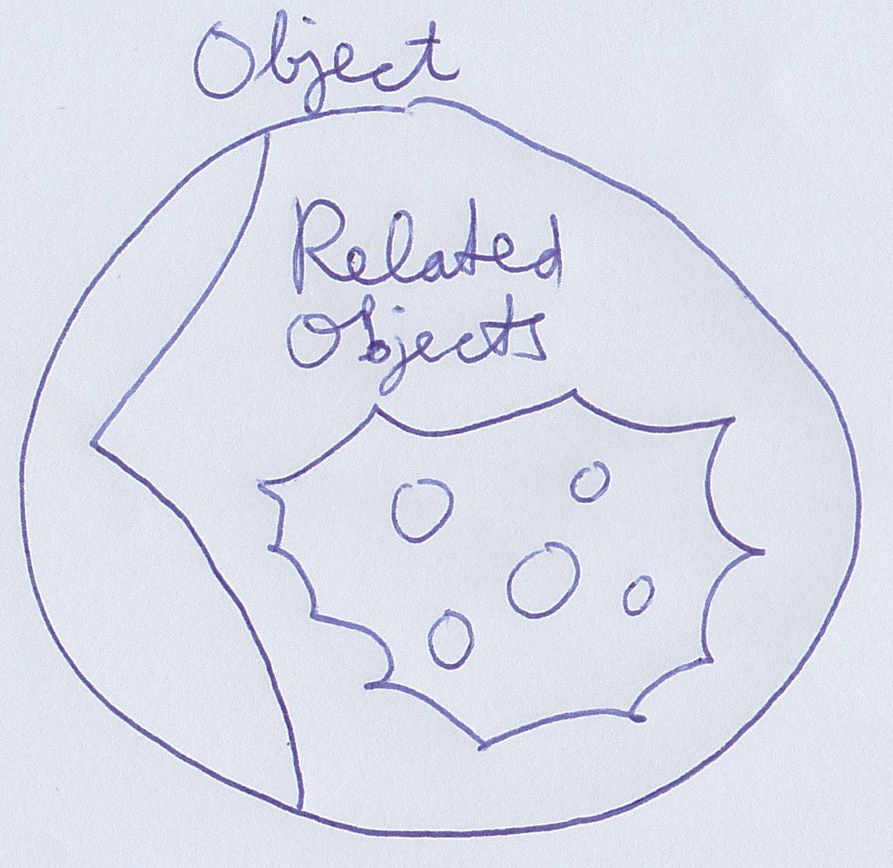


#### Reflective Data

It might just be an arbitrary choice to specifically have a *related items* collection and a *related lists* collection under the hood of a symbol. It favors distinction between **1** related item and **n** related items.

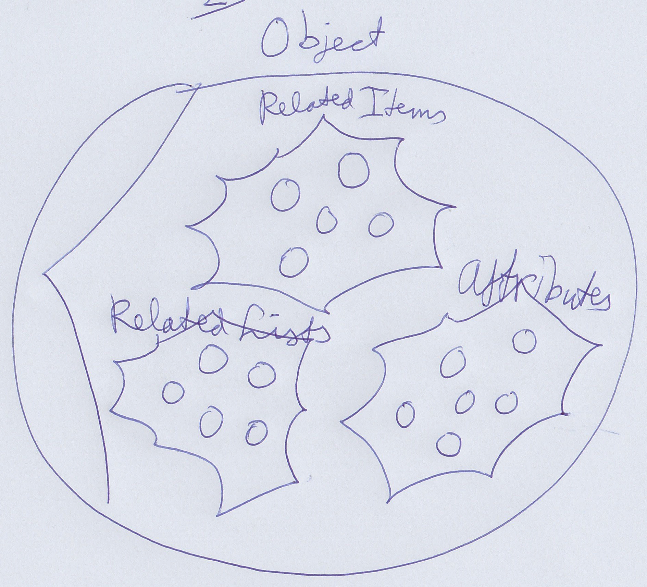
There could be variations on that.

One variation might be: having a single related objects collection:



Lists might also be in the same collection. This single related objects collection might simply contain both lists and single items.

Another variation might add another collection:



Here next to **Related Item**s, **Related Lists** there would be **Attributes**: basically a collection of simple loose values an object might hold.

It might go on. A **Commands** collection might be added for instance, to perhaps list sub-objects that are command symbols.

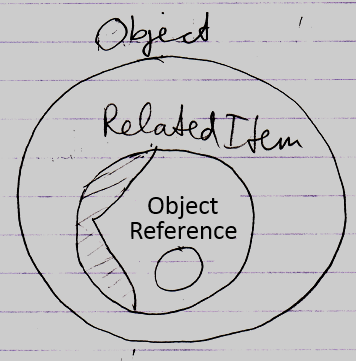
By now a suspicion might be, that it could be a quite arbitrary choice of collections.

An alternative could be that there is indeed one collection of related objects, but separate enumerators to retrieve related lists, related items, attributes and commands separately.

This 'under the hood' data might be called reflective data by some. There are other systems, where you can take a class and retrieve separate lists of properties and methods and for instance the ability to read out values or call methods. That idea might be closely related to this one here. That way you might tap into data more generically based on characteristics, rather than pointing out specific sub-objects.

#### Reference in a System Interface

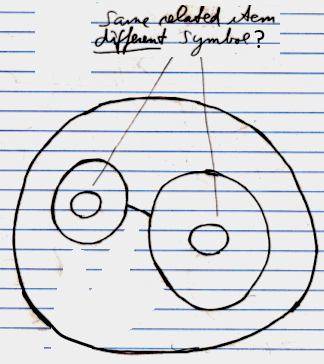
A related item might be the result of a relationship. A related item might wrap a reference to a related object.



The inner-most object reference is like a raw pointer while the related item that surrounds it likes to decorate it with info like its class and procedures around it, like a retrieval procedure.

#### Symbol

The term *symbol* mightbe used as a synonym for related item or for object. But there might be a subtle difference that a symbol might just be a shape displayed on screen. The same related item could be displayed on screen multiple times.



But the difference might be too subtle to talk about. A symbol might not be a system object, it might just a shape displayed on screen.

#### System Object

The terms above indicate different kinds of *system objects*, except for maybe the term *symbol*.

#### Summary

All the objects in this imaginary implementation of Circle might live as these system objects.

Here is a recap of terms introduced in this section:

**Object**

**Item**

**Reference**

**Object Reference**

**Pointer**

**Related Object**

**Related Item**

**Related List**

**Related List Item**

**Related Items & Related Lists Collections**

**Reflective Data**

**Symbol**

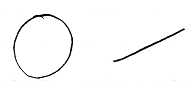
**System Object**

### System Aspects

Behavior of objects, references and lists might be controlled by controlling their *aspects*. This section tries to list out various aspects that objects, references and lists might have. The pictures try to show a bit of each aspect’s symbolization.

#### Object

The **Object** aspect may determine which object is pointed to.



#### Class

The **Class** aspect might determine which other object would function as a prototype or class of another object. It can also bind a reference to a class.



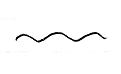
#### Interface

The **Interface** aspect might give control over how objects might look on the outside, while the insides of the objects may be different. The **Interface** aspect may be covered in the *Interfaces* chapter, and might not be mentioned any further in this chapter.



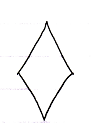
#### Value

The **Value** aspect might allow storing binary content and might allow yielding over values from one object to another.



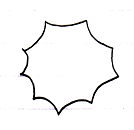
#### Execute

The **Execute** aspect might be about the ability to execute an object as a command.



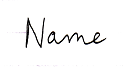
#### List

The **List** aspect may allow adding and removingitems from a list.



#### Name

The **Name** aspect may allow giving names to objects, lists and references.



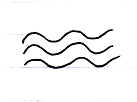
#### Existence

The **Existence** aspect might allow creating a new object. An object reference might also be annulled. Then it may point to nothing. Another option might be an ability to *check* whether an object reference "**Nothing**"or"**null**".



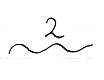
#### Data

With the data aspect the idea is to control reading or writing access. Instead of being able to access-control a single object, the **Data** aspect might control read-write access to all of the sub-objects as well.



#### Clone

The proposed **Clone** aspect might be related to the **Value** aspect, but might also copy values of sub-objects.



Its use might be that more than once there may be the desire to copy an object + its child values.

#### Summary

Here is an attempt to list the aspects again:

**Object**

**Class**

**Interface**

**Value**

**Execute**

**List**

**Name**

**Existence**

**Data**

**Clone**

### Object-Bound & Reference Bound Aspects

When using *system aspects*, there seems to be a subtle phenomenon that some aspects might to be bound to an *object*, while other aspects may be bound to a *reference*.

Object-bound aspects would be determined by an object. An object might also control its sub-objects’ reference-bound aspects. Therefore *reference-bound* aspects might also be called *sub-object-bound* aspects.

The following aspects may be object-bound:

**Value**

**Class**

**Execute**

**Clone**

**Data**

**List**

The following aspects might be reference-bound (or *sub-object-bound*):

**Object**

**Existence**

**Class**

**Name**

The **Class** aspect seems to be both object-bound as well as reference-bound. (The **Interface** aspect might be too, but that topic might be dealt with later.)

The idea of object-bound aspects might help an object protect some of its characteristics and not be different depending on which reference points to it.

An object seems to control its object-bound aspects as well as its sub-objects’ reference-bound aspects.

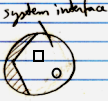
### System Interfaces & System Commands

#### System Interface Notation

Sometimes an object's normal members might be shown:



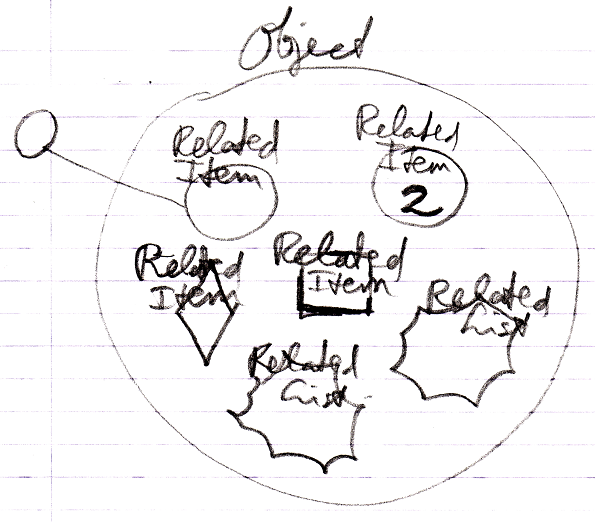
In this hypothetical Circle system, all objects might be system objects under the hood. This might include normal objects, object references and lists. They might all be system objects deeper down. An alternative to viewing the normal members like above might be for the members of a system object to be shown instead. This might be done by breaking open the inner workings of an object and show its *system interface*:



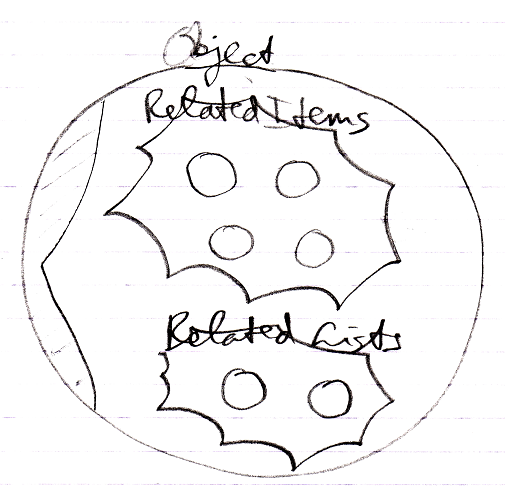
When a system interface is shown, normal members could be hidden, visible or maybe just part of them shown. The other way around, when normal members are shown, the system interface might be shown, not shown, or part visible.

##### Example: System Interface of an Object

An **Object**'s normal members might look like this:



But when the **Object**'s system interface would be opened, then for instance the **Related Items** and **Related Lists** collections might be shown instead:



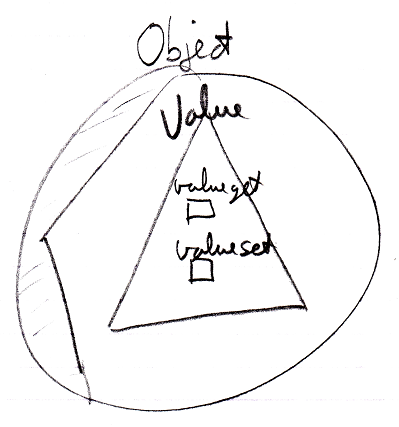
The **Related Items** and **Related Lists** mightbe displayed as circles here: regular objects, even when they are *commands* and even when they are *lists*. It might be an alternative to display items with their original shape. More than one thing is possible. This new view with system members might be a representation of the way an **Object** 'actually' internally works.

##### System Commands

System *commands* could be commands of system objects, through which aspects of objects may be controlled.

##### Aspects

Next to controlling *sub-objects*, a system interface might also control *aspects*, for instance, the **Value** aspect:



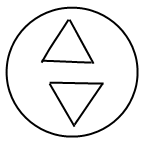
Several aspects could be given **Get** and **Set** commands. But each aspect might be controlled in a different way.

##### System Interface might be Publics of System Objects

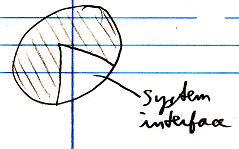
One idea that system interfaces may be based on, is that a system interface might be no more than public members of a system object. When this description here about system interfaces would be incomplete, what might be missing may be derived from the idea, that a system interface may simply show public members of a system object. Another idea derived from that could be that *private* workings of system objects might not be shown in a system interface. But these may be mere guidelines to have something to hold on to.

##### Origin of the System Interface Notation

The notation for a *system interface* may be derived from *interface* notation. When an object would have several interfaces, each interface might be shown as a triangle inside the object:

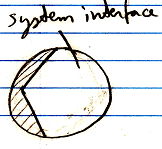


But showing a system interface would be like showing the interior of 'the symbol itself'. Therefore, the triangle of a system interface might be creatively stuck to the border of the symbol:



That way the interface might look more part of the object itself, rather than being a sub-object.

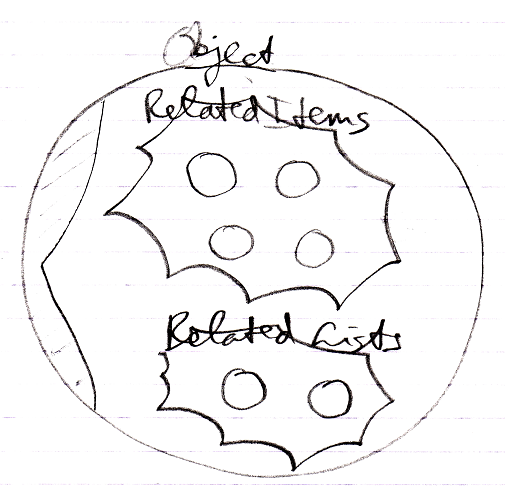
To make more room inside a system interface an exaggerated notation might be used:



It may also kind of look like the shell of a symbol is broken open and showing its inner workings. Like having opened up the system and we might see its internal wiring. It might be like seeing the setup of the symbol machine.

#### System Interface of an Object

One thing that might show when opening up a system interface of an **Object** might be **Related Items** and **Related Lists** collections.



But apart from sub-objects, an object might have the following aspects:

**Value**

**Class**

**Execute**

**Clone**

**Data**

They might be controlled through system commands. Those commands might be visible inside a system interface next to sub-object collections.

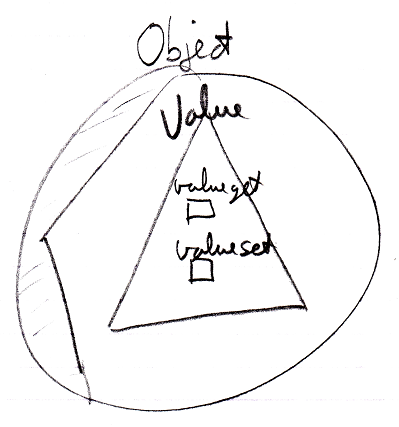
##### Value Aspect in the System Interface

The **Value** aspect might be controlled through two system commands:

**Get Value**

**Set Value**

The **Value** aspect might be represented by a triangle, that could wrap together members to control the **Value** aspect:



**Get Value** might get the value of an object.

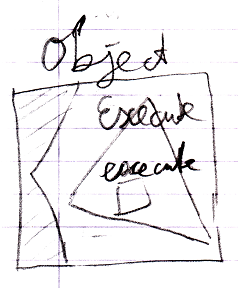
**Set Value** might set the value of an object.

##### Execute Aspect in the System Interface

The **Execute** aspect might only apply to executable objects, or 'commands'. The **Execute** aspect might be controlled through one system command:

**Execute**

The command might be placed inside a triangle, aiming to wrap together members of the **Execute** aspect:



The **Execute** command might be access controlled to prevent a command from being executed. Some commands may never be meant to execute, because they might be a definition.

It might be an idea that the **Execute** aspect might go as far as to turn a regular object into a command and back. That may ask for more system members.

##### Clone Aspect in the System Interface

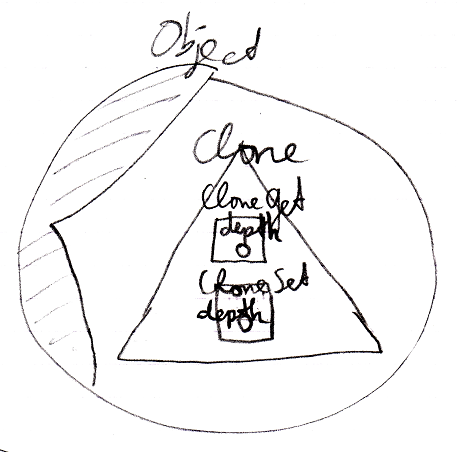
The **Clone** aspect might be controlled through two system commands:

**Get Clone**

**Set Clone**

The **Clone** aspect might be related to the **Value** aspect, but might also copy sub-objects’ values.

The commands might be placed inside a triangle, that may wrap together members of the **Clone** aspect:



**Get Clone** might copy an object and its sub-objects.

**Set Clone** mightassign cloned values to another object. This might also be a new object.

Both commands might have a parameter, that could define a cloning *depth*: the depth at which the sub-objects might be cloned. With a cloning depth of **2**, it might also be expressed as:

**Get Clone (2)**

**Set Clone (2)**

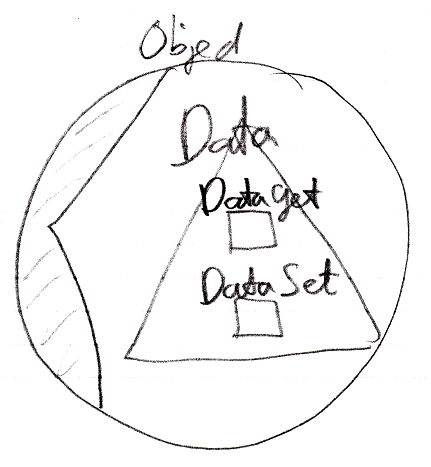
##### Data Aspect in the System Interface

The **Data** aspect might be controlled with two system commands:

**Get Data**

**Set Data**

The commands might be placed inside a triangle, that could wrap together members of the **Data** aspect:

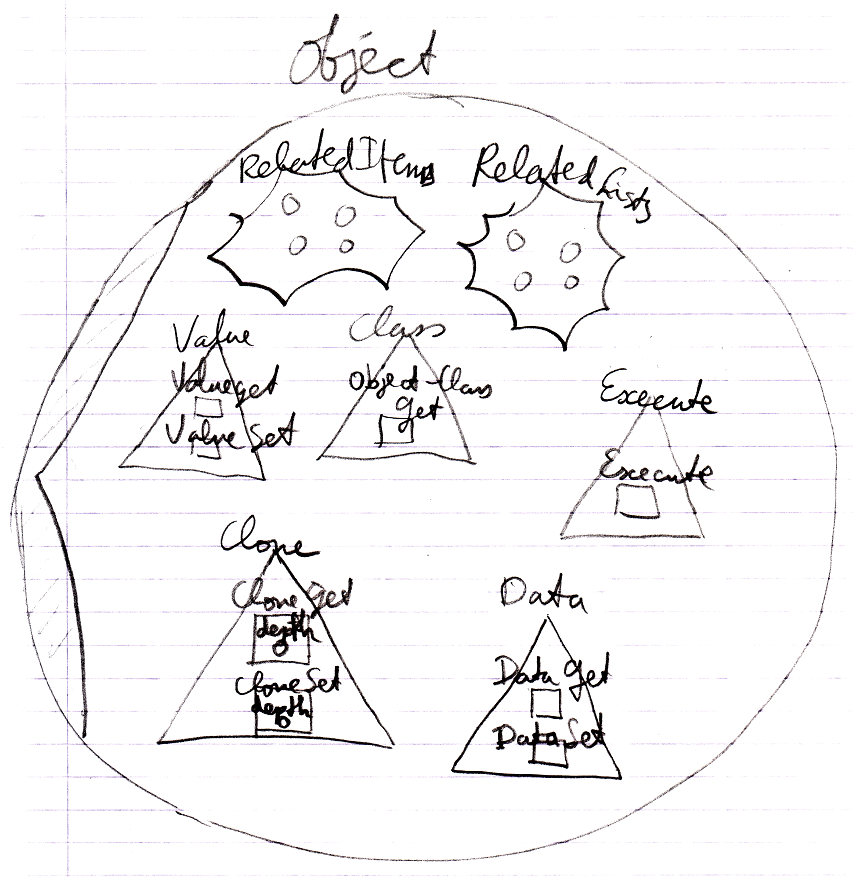


The **Get Data** and **Set Data** command might not be called. They might only be access-controlled. This might control read-write access to an object and its contents: any sub-object or deeper object.

The **Get Data** command might abstractly represent any *read*, which might mean: any possible **Get** call to any sub-object. The **Set Data** command might abstractly represent any *write* to any sub-object.

##### Overview of a System Interface for an Object

Here an attempt to visualize the system interface of an **Object** and system members introduced so far:



#### System Interface of a Related Item

When opening up the system interface for a **Related Item**, it may show system commands that might apply to **Related Items**. Aspects, that could apply to a **Related Item** may be:

**Object**

**Existence**

**Class**

**Name**

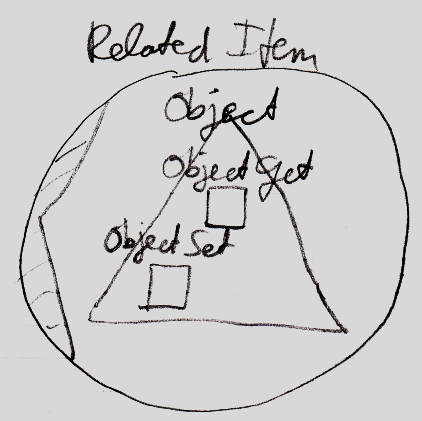
##### Object Aspect in the System Interface

The **Object** aspect of a **Related Item** might becontrolled through several system commands:

**Get Object**

**Set Object**

Commands might be placed inside a triangle, that may wrap together members of the **Object** aspect:



**Get Object** mightretrieve the targeted object of a reference.

**Set Object** maychange an object reference target.

##### Existence Aspect in the System Interface

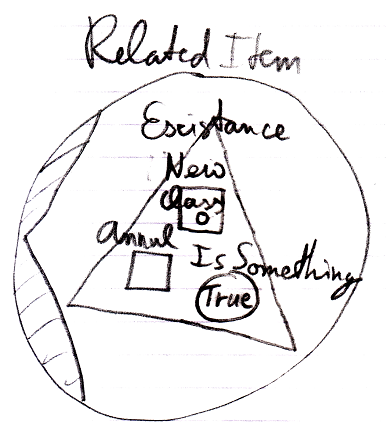
The **Existence** aspect might be controlled through the following system commands and an attribute:

**New**

**Annul**

**Is Something**

Members might be placed inside a triangle, that could wrap together members of the **Existence** aspect:



The **New** command might have an optional **Class** argument, which might indicate the class from which the object might be created.

There might also be a system attribute **Is Something** which might say **True** or **False**, which might also be represented by the terms **Something** or **Nothing**.

##### Name Aspect in the System Interface

The **Name** aspect might be controlled through two system commands:

**Get Name**

**Set Name**

The commands might be placed inside a triangle, that might wrap together members of the **Name** aspect:

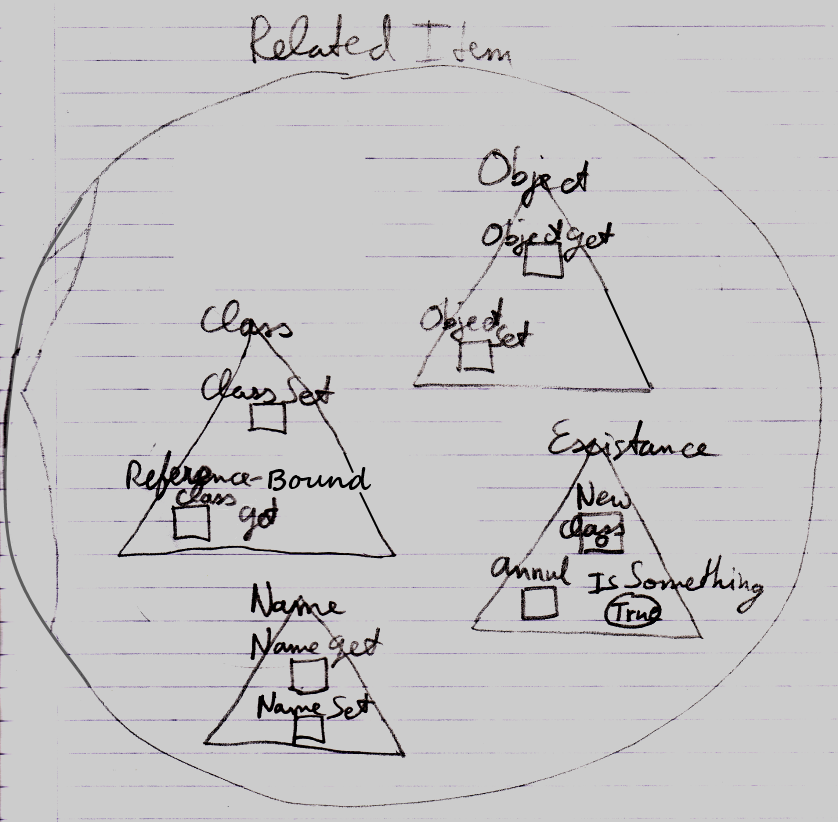


**Get Name** mightget the name of a reference or list.

**Set Name** mightset the name of a reference or list.

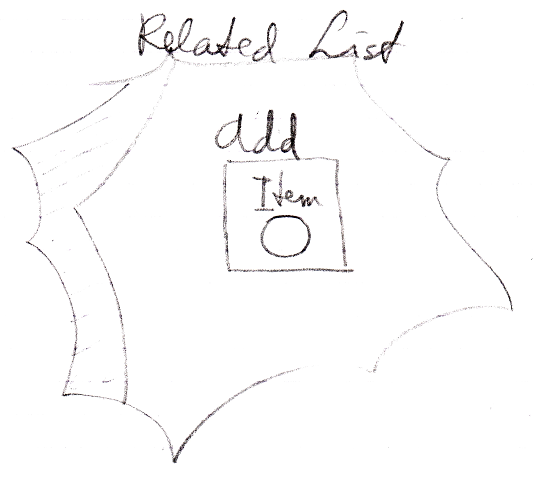
##### Overview of a System Interface for Related Item

Here an attempt to visualize the system interface of a **Related Item** and system members introduced so far:



#### System Interface of a Related List

When opening up the system interface for a **Related List**, it may show system commands applying to **Related Lists**:



The system interface of a **Related List** might show the **Add** command. The **Add** command may have an optional **Item** argument, to maybe add an existing item to a list.

But more members may be introduced later. Specifically something to desire might be to centrally fix the **Class** aspect of all the items in a list.

The **List** aspect might apply to **Related Lists**. The **List** aspect may be controlled through the following system commands:

**Add**

**Remove**

The **Add** command may be part of a **List** object. The **Add** command might add an item to a list. This may be an existing item, passed through an optional argument **Item**. A new item might also be created for it using the **New** command part of the **Existence** aspect.

The **Remove** command may be part of a **Related List Item**. It removes that item from its list.

#### System Interface of a Related List Item

When opening up the system interface for a **Related List Item**, it may show system commands applying to **Related List Items**. Aspects, that might apply to a **Related List Item** could be:

**Object**

**Existence**

**Class**

**List**

Those might be similar to a **Related Item**, minus the **Name** aspect, plus the **List** aspect. Only the **List** aspect for a **Related List Item** might be described here. Other aspects may find their descriptions where **Related** **Item** system commands were illustrated.

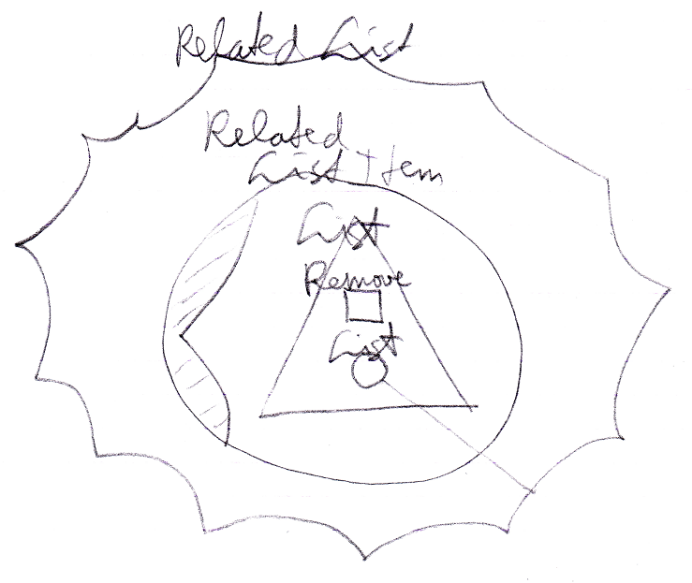
##### Related List Item’s List Aspect in the System Interface

The **List** aspect of a **Related List Item** may berepresented by the following system members:

**Remove**

**List**

Members might be placed inside an interface triangle, that might wrap together members of the **List** aspect:

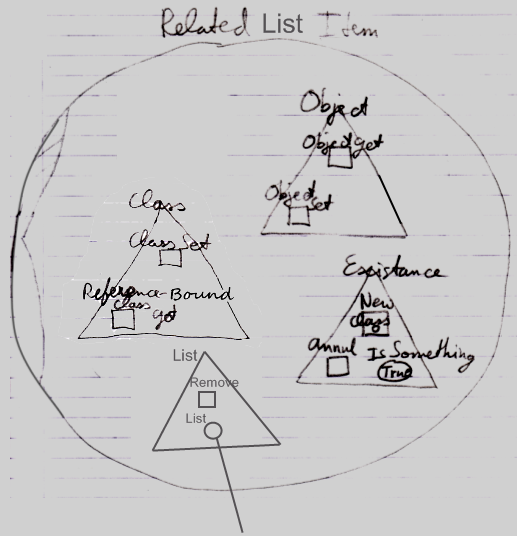


The **Remove** command might remove an item from a list.

The **List** member might be a reference to the List that a related item would be part of.

##### Overview of a System Interface for Related List Item

A **Related List Item** might be similarto a **Related Item**, so a more complete system interface of a **Related List Item** might show a similar set of aspects and might look as follows:



#### System Interface for the Class Aspect

The **Class** aspect might have a bit more to say about it than other aspects.

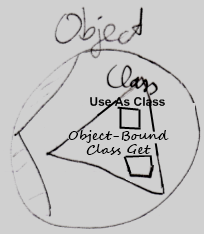
##### Object-Bound Class Aspect in the System Interface

The **Class** aspect might have several system commands, but two of them might apply to *objects*. Other ones might apply to *references*. The **Class** aspect of an **Object** might becontrolled through the following commands:

**Use As Class**

**Get Object-Bound Class**

Commands might be placed inside a triangle, that might wrap together members of the **Class** aspect:



**Use As Class** might belike **Get Object**, but then for using that object as a class of another object. This may be a common usage of the class aspect. The separate system command **Use As Class**, might also be used to separately access control whether an object might be used as a class or not.

The command **Get Object-Bound Class** may return the class object associated with an object. That command might not be used as commonly.

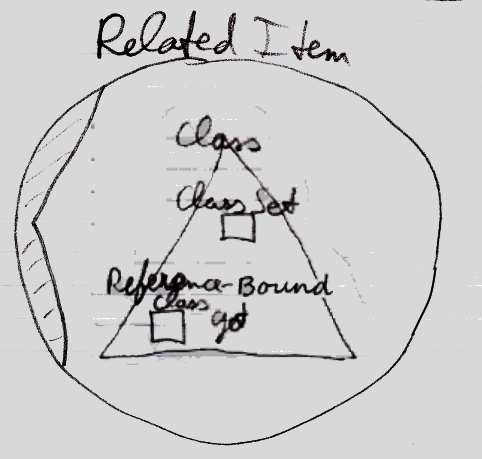
##### Reference-Bound Class Aspect in the System Interface

The **Class** aspect may have several system commands, but the following two may apply to references:

**Set Class**

**Get Reference-Bound Class**

Commands might be placed inside a triangle, that would wrap together members of the **Class** aspect:

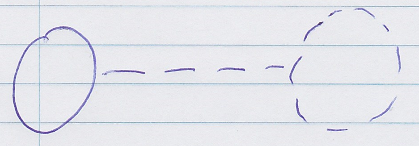


The **Set Class** command might be executed on an object reference. That reference might then only point to objects of that class. In the hypothetical system here **Set Class** mightapply only to references and possibly not to objects, because the class of an object might only be set upon creation.

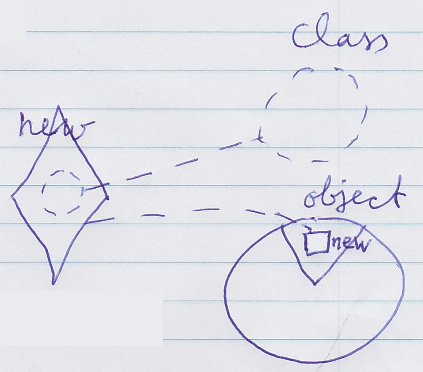
The command **Get Reference-Bound Class** may return the class object associated with the object reference. That command might not be used as commonly.

##### Class may be both Object-Bound and Reference-Bound

The **Class** aspect might apply to both objects and references, but differently. In the example system imagined here, an object could have a certain class, that might be fixed throughout its lifetime:

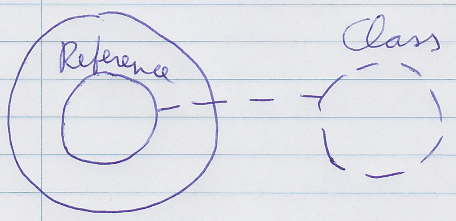


It might have beem fixed upon creation of the object:

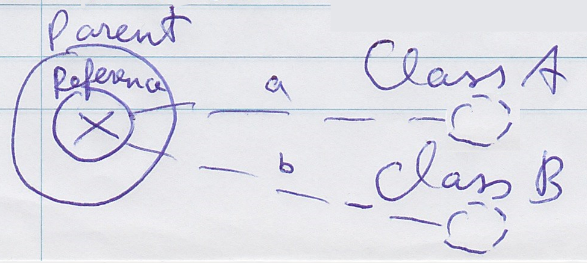


In this example system, the class of an object might never change.

A *reference* also could have a class, aiming to indicate which class of object it might point to.



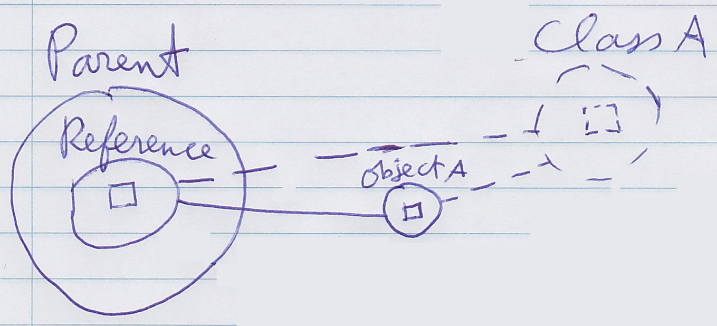
When the reference would be **Nothing** or **null**, any class might be assigned to the reference.



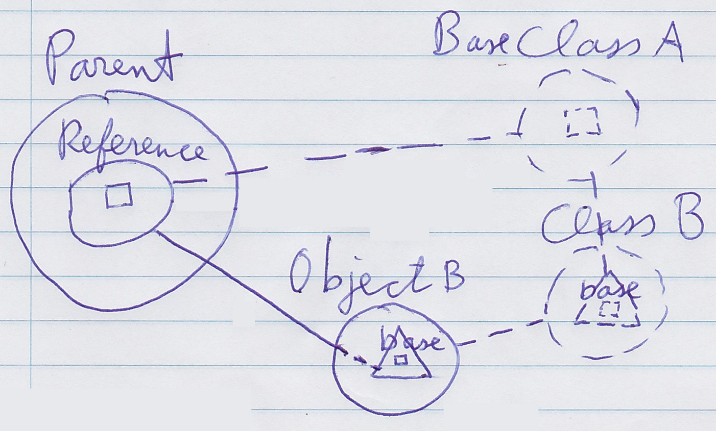
Both line **a** and **b** would both be possible.

The class of a reference mightbe changed, but only when it would be compatible with the object assigned to the reference.

The **Object-Bound Class** and **Reference-Bound Class** could point to the same class.



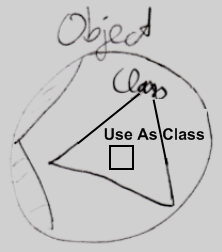
The **Reference-Bound Class** could also be a base class or an interface of the **Object-Bound Class**.



### Misc Design Choices

#### Aspect in a Triangle

An aspect might be represented by a triangle that could contain system commands:



It was a design choice to have system commands placed inside a triangle, that might represent an aspect. That way **Use As Class** for instance might be used without putting "**Class .**" or **"Object . "** in front of it.

An alternative was putting it in a circle, which would make textual alternatives of commands be something like:

**Class . Use As Class**

**Class . Use Object As**

**Object . Use As Class**

Those would all be alternatives for **Use As Class**. The idea is that this might not look as nice as just **Use As Class**.

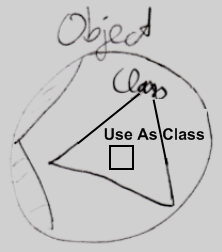
The alternative **Object . Use As Class** might look ok, but may place this class-related action inside the **Object** aspect, which might not be desirable either.

So this way **Use As Class** might be used directly instead.

This is just a design choice. It could have ended up being different.

#### 'Use' Command Gets Another Aspect

A **Use As Class** command might be made part of the **Class** aspect while it may **Get** the **Object** aspect.



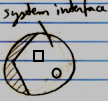
A **Use** command may **Get** a different aspect than what it applies to. A **Use As Class** command might be part of the **Class** aspect but would **Get** the **Object** aspect.

An alternative could have been to put **Use As Class** in the **Object** aspect. But **Use As Class** seems the primary use of the **Class** aspects, so to put it in the **Object** aspect might not be desirable.

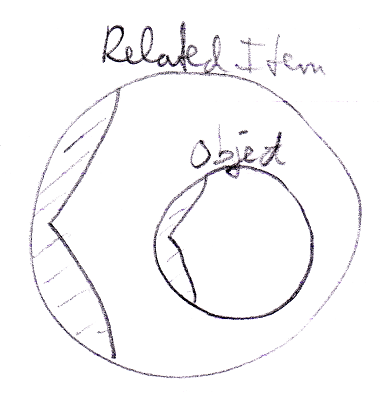
How a **Use** command might delegate to a **Get** command might not be visible in a system interface, due to its being private implementation. Only public members of a system object might be shown, not its implementation.

#### System Interfaces of Objects and References

There could be system commands that may apply to **Objects** and system commands that may apply to **References**, but when showing the system interface of a symbol, which of the two might be shown? The system interface of the **Object** or the system interface of the **Reference**?



A solution might be that it is usually *references* to objects that are dealt with, not usually an object directly, so opening up the system interface of a symbol, the system interface of the **Reference** might beshown. However, inside of it you might find the referenced **Object**, showing its system interface.



Different choices could be made here. Perhaps a distinction between **Reference** and **Object** might not be relevant for aspects that have no overlap. The **Class** aspect may be specificied separately for an **Object** and an **Reference**, for which a distinct notation might be desired. Perhaps it might be practical to keep most aspects one level deep. There are options and they might all be ok.

#### Performance Concerns?

Someone might wonder critically if Circle could run fast if running on these heavy weight system objects, instead of a regularly compiled code. One option might be that these system objects could function as a certain interpreted mode, that might compile / optimize into faster machine code. System objects might be an intermediate format of some sort. Perhaps a similar distinction as text code and machine code can be seen here.

#### Preliminariness of the System Interface Notation

The system interface notation's basics might not be preliminary. But the content of a system interface might not be precisely as specified in this documentation. One idea is that Circle might run on system objects proposed here. And that depending on how these system objects are implemented precisely, the public members of a system object could form the content of a system interface. Design choices were made in this documentation, which can influence the notation of system interface members. But hopefully it shed light on the kind of things the system interface might be used for, what roles it might play and its notation basics.